Correctly rounding elementary functions on GPU

Pierre Fortin, Mourad Gouicem and Stef Graillat

Abstract—The IEEE 754-2008 standard recommends the correct rounding of elementary functions. This requires to solve the Table Maker's Dilemma which implies a huge amount of CPU computation time. We consider in this paper accelerating such computations, namely Lefèvre algorithm, on Graphics Processing Units (GPU) which are massively parallel architectures with a partial SIMD execution (Single Instruction Multiple Data).

We first propose an analysis of the Lefèvre *hard-to-round* argument search using the concept of continued fractions. We then propose a new parallel search algorithm much more efficient on GPU thanks to its more regular control flow. We also present an efficient hybrid CPU-GPU deployment of the generation of polynomial approximations required in Lefèvre algorithm. In the end, we manage to obtain overall speedups up to 53.4x on one GPU over a sequential CPU execution, and up to 7.1x over a multi-core CPU.

Index Terms—correct rounding, Table Maker's Dilemma, Lefèvre algorithm, GPU computing, SIMD, control flow divergence, floating-point arithmetic, elementary function

1 Introduction

1.1 Problem

The IEEE 754 standard [1] specifies since 1985 the implementation of floating-point operations in order to have portable and predictable numerical software. In its latest revision in 2008 [1], it defines formats (single, double and quadruple precision), rounding modes (to the nearest and toward 0, $-\infty$ and $+\infty$) and operations ($+, -, \times, /, \sqrt{}$) returning correctly rounded values.

Furthermore, it recommends correct rounding of some elementary functions, like *log*, *exp* and the trigonometric functions. As these functions are transcendental, one cannot evaluate them exactly but have to approximate them. However, it is hard to decide which precision is required to guarantee a correctly rounded result – the rounded evaluation of the approximation must be equal to the rounded evaluation of the function with infinite precision. This problem is known as the *Table Maker's Dilemma* or TMD [2, chap. 12].

1.2 State of the art

A first strategy to solve the TMD introduced by Ziv [3] was to compute an approximation y of a function value $f(x)$ with a bounded error of ϵ (containing mathematical and round-off errors). As rounding modes are monotonic, if $y-\epsilon$ and $y+\epsilon$ rounds to the same floating-point, $f(x)$ does too : otherwise the correct rounding cannot be determined (see Fig. 1). Hence having a correctly rounded result of $f(x)$ can be done by refining the approximation y until $y-\epsilon$ and $y+\epsilon$ rounds to the same floating-point. For the most common elementary functions, such an ϵ exists according to Baker's theorem when the function is evaluated at non-zero floating-points.

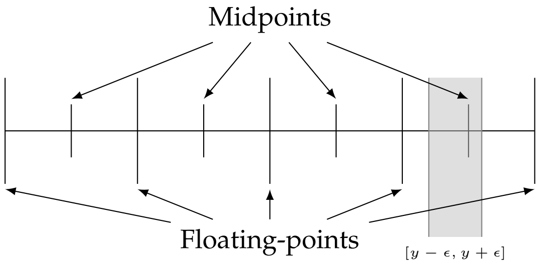


Figure 1: Example of undetermined correct rounding for rounding to nearest, where the rounding breakpoints are the midpoints of floating-point numbers.

However, the computation of many approximations can be avoided by precomputing an ϵ guaranteeing correct-rounding of the evaluation of f at any floating-point number argument. This has to be done by finding the *hardest-to-round* arguments of the function, that is to say the arguments requiring the most important precision to be correctly rounded when the function is evaluated at. These *hardest-to-round* cases can be found invoking Ziv algorithm at every floating-point number in the domain of the function, but this is prohibitive.

The first improvement was proposed by Lefèvre in [4], [5]. The main idea of his algorithm is to split the domain into several sub-domains, to "isolate" *hard-to-round* cases, and then to use Ziv algorithm to find the *hardest-to-round* cases among them. This isolation is efficiently performed using local affine approximations of the targeted function. Stehlé, Lefèvre and Zimmermann extended this method in 2003 [6], [7] (SLZ algorithm) for higher degree approximations, using the Coppersmith method for finding small roots of univariate modular

Authors are with UPMC Univ Paris 06 and CNRS UMR 7606, LIP6
Address : 4 place Jussieu, F-75252, Paris cedex 05, France
Contact: mourad.gouicem@lip6.fr

equation[1].

1.3 Motivations and contributions

Even if they are asymptotically and practically faster than exhaustive search, Lefèvre and SLZ algorithms remain very computationally intensive –around five years of CPU time for the exponential function over double precision arguments with Lefèvre algorithm for example. As both algorithms split the domain of the targeted function into sub-domains and search for *hard-to-round* cases (HR-cases) in them independently, these computations are embarrassingly parallel.

The purpose of this work is therefore to accelerate these computations on Graphics Processing Unit (GPU), which theoretically performs one order of magnitude better than CPU thanks to its massively parallel architecture. We focus here on Lefèvre algorithm, which is efficient for double precision rounding and which offers fine-grained parallelism, making it suitable for GPU.

In [8], we discussed implementation techniques to deploy Lefèvre algorithm efficiently on GPU. The major bottleneck of its deployment was the control flow divergence which is penalizing considering the partial SIMD execution (Single Instruction Multiple Data) of the GPU. Hardware [9] and software [10], [11] general solutions have been proposed recently to tackle this problem on GPU. However we here focus on algorithmic solutions.

In this paper, we thus redesign Lefèvre algorithm using the concept of continued fraction and propose a much more regular algorithm for searching HR-cases. More precisely, we strongly reduce two major sources of divergence of Lefèvre algorithm: loop divergence and branch divergence. We also propose an efficient hybrid CPU-GPU deployment of the generation of polynomial approximations using fixed multi-precision operations on GPU. These contributions enable on GPU an overall speedup of 53.4x over Lefèvre's original sequential CPU implementation, and of 7.1x over six CPU cores (with two-way SMT).

1.4 Outline

We will introduce some notions on GPU architecture and divergence in Sect. 2. We will then present in Sect. 3 some mathematical background on the Table Maker's Dilemma and properties of the set $\{a \cdot x \mod 1 \mid x < n\}$ with a fixed. In Sect. 4, we will detail the HR-case search step of Lefèvre algorithm and propose a new and more regular algorithm. We will also present their deployment on GPU. In Sect. 5 we will detail how to efficiently compute polynomial approximations on GPU. And finally, we will present performance results in Sect. 6 and conclude in Sect. 7.

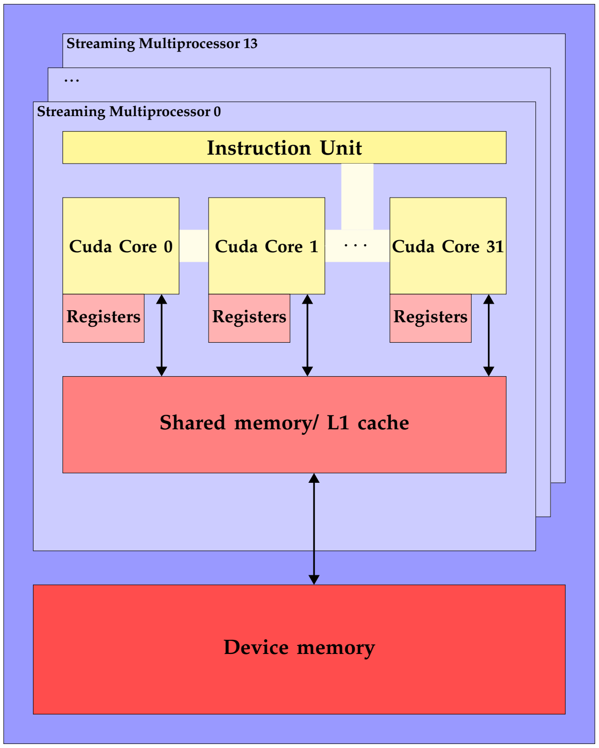


Figure 2: NVIDIA Fermi C2070 architecture.

2 GPU COMPUTING

Graphics Processing Units (GPUs) are many-core devices originally intended to graphics computations. However since mid-2000s they became increasingly used for high performance scientific computing since their massively parallel architecture theoretically performs one order of magnitude better than CPU, and thanks to the emergence of general-purpose languages adapted to GPU like CUDA [12] and OpenCL [13]. In this section we briefly describe the architecture of the latest NVIDIA GPU specialized in scientific computing (Fermi architecture), the GPU programming in CUDA and divergence problems arising from the partial SIMD execution on GPU. We use the CUDA nomenclature.

2.1 GPU architecture and CUDA programming

From a hardware point of view, a GPU is composed of several *Streaming Multiprocessors* denoted SM (14 on Fermi C2070), each being a SIMD unit (Single Instruction Multiple Data). A SM is composed of multiple execution units or *CUDA cores* (32 on Fermi) sharing the same pipeline and many registers (32768 on Fermi) as illustrated in Fig. 2. GPU memory is organized in two levels (see Fig. 2):

- *device memory*, which can be accessed by any SM on the device;
- *shared memory*, which is local to each SM.

Moreover, the device memory accesses are cached on the Fermi architecture.

From a software point of view, the developer writes in CUDA scalar code for one function designed to be

[1] http://www.loria.fr/equipes/spaces/slz.en.html

executed on the device, namely a *kernel*. At runtime, many *threads* are created by *blocks* and bundled into a *grid* to run the same kernel concurrently on the device. Each block is assigned to a SM. Within each block, threads are executed by groups of 32 called *warps*. The ratio of the number of resident warps (number of warps a SM can process at the same time) to the maximum number of resident warps per SM is named the *occupancy*. In order to increase the occupancy the number of blocks and their size have to be tuned.

2.2 Divergence

As threads are executed by warps on the GPU SIMD units, applications should have regular patterns for memory access and control flow.

Regularity of memory accesses patterns is important to achieve high memory throughput. As the threads within a warp load data from memory concurrently, the developer has to coalesce device memory accesses and avoid bank conflict in shared memory [14, chap. 6]. This can be done by reorganizing data storage.

Regularity of control flow is important to achieve high instruction throughput, and is obtained when all the threads within a warp execute the same instruction concurrently [14, chap. 9]. In fact, when the threads of a same warp diverge (i.e. they follow different execution paths), the different execution paths are serialized. For an *if* statement, the *then* and *else* branches are serially executed. For a loop, any thread exiting the loop has to wait until all the threads of its warp exit the loop. In the following we will distinguish branch divergence due to *if* statements and loop divergence due to loop statements.

The impact of branch divergence can be statically estimated by counting the number of instructions issued within the scope of the *if* statement. Let consider the *then* branch issues n_{then} instructions and the *else* branch issues n_{else} instructions.

- If the warp does not diverge, either n_{then} or n_{else} instructions are issued depending on the evaluation of the condition.
- If the warp diverges, $n_{then} + n_{else}$ instructions are issued.

Contrary to branch divergence, measuring the impact of loop divergence requires a dedicated indicator and profiling. We introduced in [8] the mean deviation to the maximum of a warp. This indicator is similar to the standard deviation, which is the mean deviation to the mean value. However, as the number of loop iterations issued for a warp is equal to the maximum number of loop iterations issued by any thread within the warp, it is relevant to consider the mean deviation to the maximum value. This gives the mean number of loop iterations a thread remains idle within its warp.

More formally, we denote ℓ_i the number of loop iterations of the thread i and we number the threads within a warp from 1 to n, n being the number of threads by warp ($n = 32$ on Fermi). If $\ell = \{\ell_i, i \in [1, n]\}$, the Mean Deviation to the Maximum (MDM) of a warp is defined as

$$\mathrm{MDM}(\ell) = \max(\ell) - \mathrm{mean}(\ell). \tag{1}$$

We can normalize the mean deviation to the maximum by $\max(\ell)$ to compute the average percentage of loop iterations for which a thread remains idle within its warp. Hence, the Normalized Mean Deviation to the Maximum (NMDM) is

$$\mathrm{NMDM}(\ell) = 1 - \frac{\mathrm{mean}(\ell)}{\max(\ell)}. \tag{2}$$

3 Mathematical preliminaries

In this section we give some definitions to introduce more formally the Table Maker's Dilemma as in [15]. We also recall some known properties and their proofs on the distribution of the elements of the set $\{a \cdot x \mod 1 \mid x < n\}$ with a fixed [5], [16]–[19].

3.1 The Table Maker's Dilemma

Before defining the Table Maker's Dilemma, we introduce some notations and definitions similar to those of [15]. We denote $\{X\}$ or $X \mod 1$ the fractional part of X. We write $X \operatorname{cmod} 1$ the centered modulo, which is the real Y such that $X - Y \in \mathbb{Z}$ and $Y \in]-1/2, 1/2]$ (Y equals $X - \lfloor X \rfloor$ or $X - \lceil X \rceil$ depending on which has the lowest absolute value). We also write $\#_p E$ the number of precision-p floating-point numbers in the set E.

Definition 1. *The mantissa $m(x)$ and exponent $e(x)$ of a non-zero real number x are defined by $|x| = m(x) \cdot 2^{e(x)}$ with $1/2 \leq m(x) < 1$.*

Definition 2. *We define*

$$\mathrm{dist}_p(x) = |2^p \cdot m(x) \operatorname{cmod} 1|$$

as the distance between a real number x and the closest precision-p floating-point number.

Definition 3. *We now define a (p, ϵ)* hard-to-round case *(or* HR-case*) of a real-valued function f as a precision-p floating-point number x solution of the inequality*

$$\mathrm{dist}_p(f(x)) < \epsilon. \tag{3}$$

It has to be noticed that if x satisfies equation (3), it also satisfies

$$2^p \cdot m(f(x)) + \epsilon < 2\epsilon \mod 1. \tag{4}$$

Equation 4 is used to test if an argument is a (p, ϵ) HR-case, avoiding the computation of absolute values and cmod.

Hence, a $(p, 2^{-p'})$ HR-case x is a precision-p floating-point number for which $f(x)$ is at a distance less than $2^{-p'}$ from the closest precision-p floating-point number. In other words, more than $p + p'$ bits are necessary to correctly round $f(x)$ at precision p.

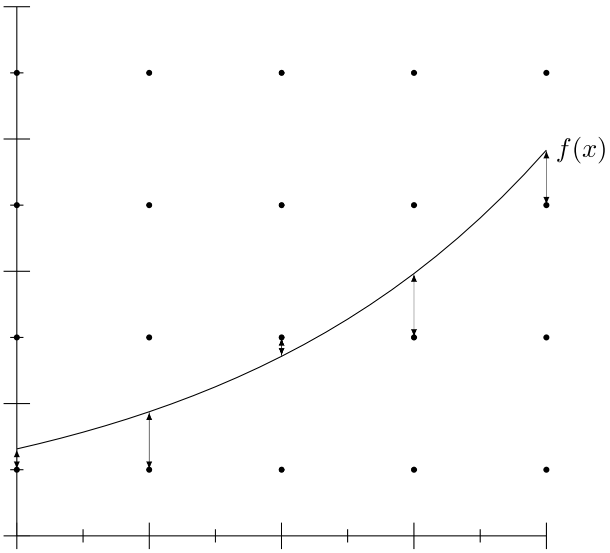


Figure 3: Distances between the curve defined by f and the rounding breakpoints for rounding-to-nearest.

The given definition of HR-case only applies for directed rounding. However, this definition can be extended to all IEEE-754 rounding modes as rounding-to-nearest (p, ϵ) HR-cases are directed rounding $(p+1, 2\epsilon)$ HR-cases.

Definition 4 (Table Maker's Dilemma). *If f is a real valued function defined over a domain D, we define the Table Maker's Dilemma as finding an ε such that*

$$\mathrm{dist}_p(f(x)) > \varepsilon$$

for any precision p floating-point number $x \in D$.

We call *hardest-to-round* cases the arguments $x \in D$ minimizing $\mathrm{dist}_p(f(x))$. Knowing the hardest-to-round cases gives us a lower bound on the distances between the function f and the rounding breakpoints (see Fig. 3) and therefore a solution to the TMD.

The general method to find the hardest-to-round case of a function is the following:

1) fix a "convenient" ϵ using probabilistic assumptions [2, Sect. 12.2],
2) find (p, ϵ) HR-cases with *ad hoc* methods such as Lefèvre or SLZ algorithms,
3) find the hardest-to-round among the (p, ϵ) HR-cases using Ziv method [3].

The most compute intensive step in this method is the second one. Lefèvre or SLZ algorithms both relies on the following two major steps.

1) *Generation of polynomial approximations*: given ϵ_{approx}, approximate function f by polynomials P_i on sub-intervals D_i such that $|P_i(x) - f(x)| < \epsilon_{approx}2^{p+e(f(x))}$ for all $x \in D_i$.
2) *HR-case search*: find the (p, ϵ') HR-cases of P_i with $\epsilon' = \epsilon + \epsilon_{approx}$ which are the (p, ϵ) HR-cases for f in D_i.

In the HR-case search of both algorithms, a Boolean test is used to isolate HR-cases. It successes if there is no (p, ϵ') HR-case for P_i in D_i and fails otherwise.

In this paper, we focus on Lefèvre algorithm which truncates polynomials to degree one for Boolean test. We simplify the notations by considering $D_i = [X_0, X_n]$ with $e(X) = e(X_0)$ for all floating-point number X in D_i. We also denote $Q_i(X) = P_i(X) \mod X^2$ the truncation of P_i to degree one with $|Q_i(X) - P_i(X)| < \epsilon_{trunc}2^{p+e(P_i(X))}$, and

$$2^p \cdot m(Q_i(X)) + \epsilon'' = b - a \cdot x,$$

with $x = 2^{p-e(X_0)}(X - X_0)$ and $\epsilon'' = \epsilon' + \epsilon_{trunc}$. This change of variable enables us to consider the integers $x \in [\![0, \#_p D_i]\!]$ instead of the precision-p floating-point numbers $X \in D_i$. Hence, the Boolean test of Lefèvre algorithm consists of testing if the following inequality holds:

$$\inf\{b - a \cdot x \mod 1 \mid x < \#_p D_i\} < 2\epsilon''. \tag{5}$$

More precisely, if the inequality (5) does not hold, there is no $(p, \epsilon + \epsilon_{approx} + \epsilon_{trunc})$ HR-cases for Q_i in D_i which implies there is no $(p, \epsilon + \epsilon_{approx})$ HR-case for P_i in D_i.

Moreover, we remark that computing the infimum of the set $\{b - a \cdot x \mod 1 \mid x < \#_p D_i\}$ is similar to find the multiple of a which is the closest to the left of b modulo 1 on the unit segment.

3.2 Properties of the set $\{a \cdot x \mod 1 \mid x < n\}$

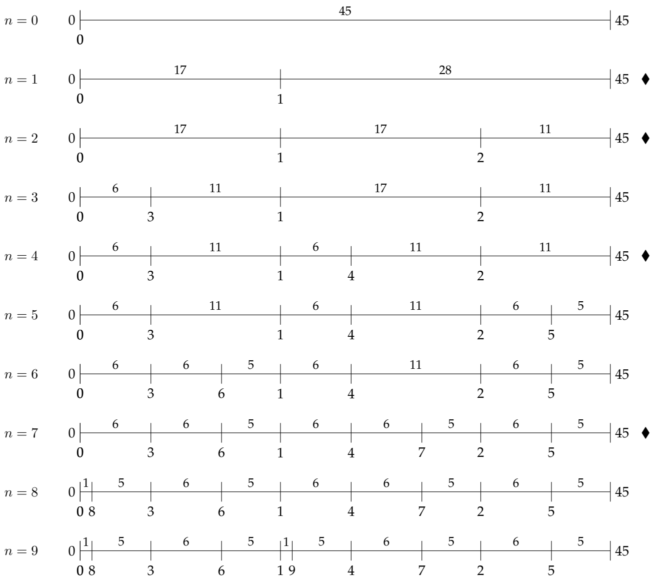
Here we will detail some properties on the points $\{a \cdot x \mod 1 \mid x < n\}$ over the unit segment. These properties are necessary to efficiently locate the closest point to $\{b\}$ in these configurations.

Theorem 1 (Three distance theorem). *Let a be an irrational number. If we place on the unit segment $[0, 1[$ the points $\{0\}$, $\{a\}$, $\{2a\}$, $\ldots$, $\{(n-1)a\}$, these points partition the unit segment into n intervals having at most three lengths with one being the sum of the two others.*

Theorem 1 is also known as the Steinhaus, the three length or the three gap theorem. It has first been proved by Slater [16]. Different approaches to prove this theorem can be found in [17], [18] and surveys of these methods in [19], [20].

Hereafter, we give a short description of how are constructed the configurations $\{a \cdot x \mod 1 \mid x < n\}$ when increasing n, and some properties over these configurations. Both are widely based on Van Ravenstein [19] and Lefèvre [21] proofs. The construction of the configurations $\{a \cdot x \mod 1 \mid x < n\}$ is illustrated in Fig. 4.

1) **Hypothesis**
   a is irrational.
2) **Base case** $n = 2$
   Only two points are on the segment, $\{0\}$ and $\{a\}$. As $\{a\} < 1$ by definition, we obtain the configuration $(a, 1 - a)$ with the leftmost interval of length a and the rightmost interval of length $1 - a$.



$n=0$ 0 45 45 0
$n=1$ 0 17 28 45 ♦ 0 1
$n=2$ 0 17 17 11 45 ♦ 0 1 2
$n=3$ 0 6 11 17 11 45 0 3 1 2
$n=4$ 0 6 11 6 11 11 45 ♦ 0 3 1 4 2
$n=5$ 0 6 11 6 11 6 5 45 0 3 1 4 2 5
$n=6$ 0 6 6 5 6 11 6 5 45 0 3 6 1 4 2 5
$n=7$ 0 6 6 5 6 6 5 6 5 45 ♦ 0 3 6 1 4 7 2 5
$n=8$ 0 1 5 6 5 6 6 5 6 5 45 0 8 3 6 1 4 7 2 5
$n=9$ 0 1 5 6 5 1 5 6 5 6 5 45 0 8 3 6 1 9 4 7 2 5

Figure 4: Computation of the $\{a \cdot x \mod 1 \mid x < n\}$ for $0 \leq n < 10$, with $a = 17/45$. Two-length configurations are marked with ♦.

3) **Inductive step**
Given the configuration $\{a \cdot x \mod 1 \mid x < n\}$, let consider the leftmost interval is of length p and the rightmost interval is of length q.
To build $\{a \cdot x \mod 1 \mid x < n+1\}$ from $\{a \cdot x \mod 1 \mid x < n\}$, we add the point $\{na\}$ on the segment. Adding the point $\{na\}$ splits an interval as a is irrational (otherwise we would have $na = ka \mod 1$ which contradicts the hypothesis on the irrationality of a).
Let denote $\{la\}$ (respectively $\{ra\}$) the closest point to the left (resp. right) of $\{na\}$ in $\{a \cdot x \mod 1 \mid x < n\}$.

   - If $l \neq 0$ and $r \neq 0$:
     $\{la\}$, $\{ra\}$ and $\{na\}$ admit $\{(l-1)a\}$, $\{(r-1)a\}$ and $\{(n-1)a\}$ as preimages by adding $-a$ on the segment. Hence, the distance between $\{la\}$ and $\{na\}$ equals the distance between $\{(l-1)a\}$ and $\{(n-1)a\}$ and the distance between $\{na\}$ and $\{ra\}$ equals the distance between $\{(n-1)a\}$ and $\{(r-1)a\}$. That is to say, by placing the point $\{na\}$, we split an interval in the same way $\{(n-1)a\}$ did.
   - If $l \neq 0$ and $r = 0$:
     first it has to be noticed that 0 has no preimage on the segment because if it has one, there may exist $k \neq 0$ such that $a \cdot k = 0 \mod 1$, which contradicts that a is irrational. Then, according to the three distance theorem, there are at most three lengths of interval on the segment, namely $(p, q, q-p)$ or $(p, q, p-q)$. As 0 is the right endpoint of the interval $[\{la\}, \{ra\}]$, this interval is the rightmost one and its length is q by hypothesis. Then, by adding $\{na\}$, we split the rightmost interval, of length q, in two intervals of length p and $q-p$ and the length $q-p$ is new as 0 has no preimage. Moreover, as we have created an interval of a third length by adding the point $\{na\}$, this implies there were only two lengths on the segment before adding $\{na\}$ and that $p < q$.
   - If $l = 0$ and $r \neq 0$:
     by the same argument, according to the three distance theorem, there are at most three lengths of interval on the segment, namely $(p, q, q-p)$ or $(p, q, p-q)$. As 0 is the left endpoint of the interval $[\{la\}, \{ra\}]$, this interval is the leftmost one and its length is p by hypothesis. Hence, we have just split an interval of length p in two intervals of length $p-q$ and q and the length $p-q$ is new as 0 has no preimage on the segment. Moreover, as we have created an interval of a third length by adding the point $\{na\}$, this implies there were only two lengths on the segment before adding $\{na\}$ and that $q < p$.
   - If $l = 0$ and $r = 0$:
     this cannot happen if $n > 2$.

We now introduce three properties on how the points are added on the configurations $\{a \cdot x \mod 1 \mid x < n\}$. Property 1 is illustrated on an example in Fig. 4 and properties 2 and 3 in Fig. 5.

Property 1. *Let (p, q) be a two-length configuration. Let denote $h = max(p, q)$ and $l = min(p, q)$. The next two-length configuration is $(l, h-l)$.*

Proof: Before splitting an interval having 0 as endpoint (see cases with either $r = 0$ or $l = 0$ above) we have a two-length configuration. When splitting an interval not having 0 as endpoint (see case with $l \neq 0$ and $r \neq 0$ above), we split this interval in the same way than the previous one was.

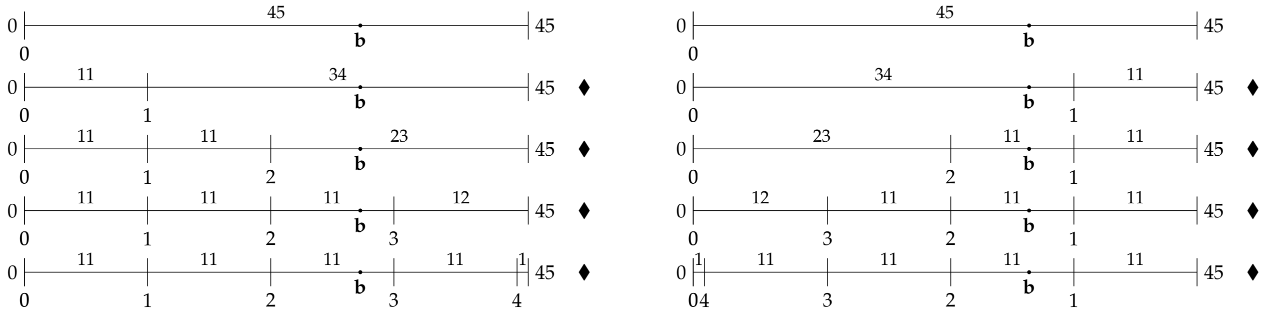
When splitting an interval having 0 as left endpoint, we split an interval p into two intervals $p-q$ and q. Splitting all intervals of length p into two intervals $p-q$ and q leads to the next two-length configuration with lengths $(p-q, q)$.

When splitting an interval having 0 as right endpoint, we split an interval q into two intervals p and $q-p$. Splitting all intervals of length q into two intervals p and $q-p$ leads to the next two-length configuration with lengths $(p, q-p)$. □

Property 2. *Let (p, q) be a two-length configuration. Let denote $h = max(p, q)$ and $l = min(p, q)$. If we denote $r = h - \lfloor h/l \rfloor \cdot l$ the remainder of the division of h by l, then (l, r) is a two-length configuration.*

Proof: It is similar as repeating $\lfloor h/l \rfloor$ times the Property 1. □

Property 1 and 2 implies that computing the lengths of the two-length configurations is similar to compute the continued fraction expansion of a, where all the



(a) $a = 11/45$; $b = 30/45$ (b) $a = 34/45$; $b = 30/45$

Figure 5: Computation of $\{a \cdot x \mod 1 \mid x < n\}$ for $0 \leq n < 5$ illustrating the directed reduction Property. Two-length configurations are marked with ♦.

lengths of the two-length configurations correspond to the partial remainders [19], [20].

Property 3 (Directed reduction). *Let (p, q) be a two-length configuration with $p < q$ (resp. $q < p$). When going to the next two-length configuration, intervals of length q (resp. p) are split into two intervals: a left one of length p (resp. $p - q$) and a right one of length $q - p$ (resp. q).*

Moreover, if we denote $r = q - \lfloor q/p \rfloor \cdot p$ the remainder of the division of q by p (resp. $r = p - \lfloor p/q \rfloor \cdot q$ the remainder of the division of p by q), intervals of length q (resp. p) are split into k intervals of length p and one interval of length r (resp. one interval of length r and k intervals of length q) in this order i.e. from left to right on the unit segment (resp. from right to left on the unit segment).

Proof: When adding a point in an interval having 0 as endpoint, a new interval with a new distance is created with 0 as endpoint. Hence, when the leftmost interval of length p is split into two intervals $p - q$ and q, the interval of length $p - q$ is the new leftmost interval. Similarly when the rightmost interval of length q is split into two intervals p and $q - p$, the interval of length $q - p$ is the new rightmost interval.

Applying this argument to the construction of Property 2 proves Property 3. □

All the proofs provided in this section are valid as 0 has no preimage when a is irrational. However, this is also valid for a rational as long as neither p nor q equals to 0 (that is to say, the last quotient of the continued fraction expansion is computed).

4 HR-case search on GPU

In this section we describe two algorithms for HR-case search: Lefèvre HR-case search and a new and more regular HR-case search. More precisely if P_i is a polynomial of degree δ defined over a domain D_i, they find every precision p floating-point number $x \in D_i$ satisfying (4) for $P_i(x)$.

4.1 Lefèvre HR-case search

In [21], Lefèvre presented an algorithm to search for (p, ϵ') HR-cases of a polynomial $P_i(x)$. This algorithm relies on a Boolean test which truncates $P_i(x)$ to degree one, computes a lower bound of the set $\{b - a \cdot x \mod 1 \mid x < \#_p D_i\}$ and checks if the inequality (5) holds.

In Sect. 3.2, we described some properties of the configurations $\{a \cdot x \mod 1 \mid x < n\}$. According to these properties, computing the lengths of the intervals of the two-length configurations can be done efficiently in $O(\log \#_p D_i)$ arithmetic operations by computing the continued fraction expansion of a. However, if we use continued fraction expansion, we will place more points than $\#_p D_i$ on the unit segment (at most $2 \cdot \#_p D_i$ if we use the subtraction-based Euclidean algorithm). To take advantage of the efficient construction of the two-length configurations, Lefèvre HR-case search computes the infimum of $\{b - a \cdot x \mod 1 \mid x < n\}$ with $n > \#_p D_i$ the number of multiples of a placed, which gives a lower bound on $\{b - a \cdot x \mod 1 \mid x < \#_p D_i\}$. Then the infimum of $\{\text{dist}_p(P_i(x)) < \epsilon \mid x < \#_p D_i\}$ is exactly computed by exhaustive search (only if required) in $O(\#_p D_i)$ arithmetic operations. To minimize this exhaustive search, a filtering strategy in three phases is used.

- Phase 1: we compute a lower bound on $\{b - a \cdot x \mod 1 \mid x < \#_p D_i\}$ and test if this lower bound is a (p, ϵ'') HR-case of Q_i. If not there is no (p, ϵ') HR-case for P_i in D_i. Else, go to next phase.
- Phase 2: we split D_i in sub-intervals $D_{i,j}$, we refine the approximation $Q_i(x)$ by $Q_{i,j}(x)$ and we compute a lower bound on $\{b_j - a_j \cdot x \mod 1 \mid x < \#_p D_{i,j}\}$ for each $D_{i,j}$. For each $D_{i,j}$ where the lower bound on $\{b_j - a_j \cdot x \mod 1 \mid x < \#_p D_{i,j}\}$ is a (p, ϵ''_j) HR-case of $Q_{i,j}$, go to next phase.
- Phase 3: we search exhaustively for (p, ϵ') of P_i in $D_{i,j}$ using the table difference method (see Sect. 5).

The corner stone of Lefèvre algorithm strategy is therefore the computation of the infimum of $\{b - a \cdot x \mod 1 \mid x < n\}$. In other words, it finds which multiple of a is the closest to the left of b modulo 1.

Hereafter we focus on computing the distance between $\{b\}$ and the closest point to the left of $\{b\}$, denoted $\{la\}$, in the configuration $\{a \cdot x \mod 1 \mid x < n\}$. We write N the number of floating-point numbers in the considered interval ($n \geq N$ as we compute a lower bound). Depending on how we generate the two-length configurations (using Property 1 or Property 2) we can derive from Property 3 two ways to compute the distance between $\{b\}$ and $\{la\}$.

Algorithm 1: Lefèvre lower bound computation and test algorithm.

```
   input : $b - a \cdot x$, $\epsilon''$, $N$
 1 initialisation: $p \leftarrow \{a\}$;  $q \leftarrow 1 - \{a\}$;  $d \leftarrow \{b\}$;
                   $u \leftarrow 1$;  $v \leftarrow 1$;
 2 if $d < \epsilon''$ then return Failure;
 3 while True do
 4     if $d < p$ then
 5         $k = \lfloor q/p \rfloor$;
 6         $q \leftarrow q - k * p$; $u \leftarrow u + k * v$;
 7         if $u + v \geq N$ then return Success;
 8         $p \leftarrow p - q$; $v \leftarrow v + u$;
 9     else
10         $d \leftarrow d - p$;
11         if $d < \epsilon''$ then return Failure;
12         $k = \lfloor p/q \rfloor$;
13         $q \leftarrow p - k * q$; $v \leftarrow v + k * u$;
14         if $u + v \geq N$ then return Success;
15         $q \leftarrow q - p$; $u \leftarrow u + v$;
```

In the lower bound computation of Lefèvre HR-case search, the way we compute the two-length configurations depends on the length of the interval containing $\{b\}$. When adding points in intervals containing $\{b\}$ and in the direction of $\{b\}$ we use a subtraction-based Euclidean algorithm (Property 1). Else we use a division-based Euclidean algorithm (Property 2). Algorithm 1 describes the lower bound computation of Lefèvre HR-case search and the corresponding test with respect to ϵ'' which is the sum of all errors involved. In this algorithm, the variables u and v count the number of intervals of length q and p respectively, in order to exit when $n = u+v > N$. They are updated by maintaining $pu+qv = 1$. The variable d contains the distance between $\{b\}$ and the closest point to its left $\{la\}$. Hereafter we detail the relations between the two-length configurations and the execution paths of Algorithm 1. Let (p, q) be a two-length configuration.

- If $p < q$:
  - if $\{b\}$ is in an interval of length p, no point is added in the interval containing $\{b\}$. So we go directly to the configuration (p, r) with $r < p$ and $r = q - \lfloor q/p \rfloor \cdot p$.
  - if $\{b\}$ is in an interval of length q and $d < p$, as only intervals of length q can be split by intervals of length p, if $d < p$ no points can be added to the left of b. So we go directly to the configuration (p, r) with $r < p$ and $r = q - \lfloor q/p \rfloor \cdot p$.
  - if $\{b\}$ is in an interval of length q and $d > p$, we subtract p to d and we go to the configuration $(p, q - p)$.
- If $q < p$:
  - if $\{b\}$ is in an interval of length q, no point is added in the interval containing $\{b\}$. So we go directly to the configuration (r, q) with $r < q$ and $r = p - \lfloor p/q \rfloor \cdot q$.
  - if $\{b\}$ is in an interval of length p and $d < p$, according to Property 3 points are added from the right in the intervals of length p. As $d < p$, $\{b\}$ is in an interval of length p and points can potentially be added to the left of $\{b\}$. Hence we go to configuration $(p - q, q)$.
  - if $\{b\}$ is in an interval of length p and $d > p$, according to Property 3 points are added from the right in the intervals of length p. As points at the right of $\{b\}$ have been added one by one in the previous rule, we are sure that the last point added is the closest point at the left of $\{b\}$ added so far. Hence we refresh d by subtracting p. As $\{b\}$ is no more in an interval of length p, we go directly to configuration (r, q) with $r < q$ and $r = p - \lfloor p/q \rfloor \cdot q$.

It has to be noticed that the condition at line 4 is true if $\{b\}$ is in an interval of length p and false otherwise. This condition enables thus to handle the six previous cases.

It also has to be noticed that Lefèvre algorithm always reduces d by using subtractions at line 10. In practice Lefèvre adds specific instructions to compute partly these reductions with divisions in order to avoid large quotients to be entirely computated with subtractions. We have omitted these instructions here for clarity but they are present in our implementations of Lefèvre algorithm.

Furthermore as the algorithm computes a continued fraction, the remainders have to be computed at each iteration. In practice, we can make use of different division implementations to compute these remainders. We can apply a subtractive division, a division instruction, or combine both in an hybrid approach as presented in [21].

Let write C_{div} (resp. C_{sub}) the cost of the division instruction (resp. the subtraction instruction). The subtractive division cost is $q \cdot C_{sub}$ with q the computed quotient. The cost of the division instruction is constant and equals C_{div}. Then, if the computed quotient is less than $\frac{C_{div}}{C_{sub}}$, subtractive division is more efficient than division instruction, else division instruction is more efficient than subtractive division.

Lefèvre hybrid division consists in choosing the best division implementation each time we compute a quotient. As we cannot rely on the quotient itself as a criteria, the expected size of the quotient is used. If we divide a by b, the size of the expected quotient can be estimated by the difference of size between a and b. Let k be a threshold on the size of the expected quotient. If $a > 2^k b$ (namely, a is rather big compared to b) we use the division instruction, else we use the subtractive division.

However, in our context, we expect small quotients as the quotients of continued fraction expansions follow the Gauss-Kuzmin distribution [22], [23] [24, p. 352].

Theorem 2 (Gauss-Kuzmin)**.** *Given a continued fraction* $[0, a_1, a_2, \ldots]$ *and* $k \in \mathbb{N}^*$,

$$\lim_{n\to\infty} \mathbb{P}(a_n = k) = -\log_2 \left(1 - \frac{1}{(k+1)^2}\right).$$

Even if the threshold k is likely small, setting it to a relevant value directly depends on the architecture and is determined by extensive testings.

4.2 New regular HR-case search

We here propose a new algorithm for the HR-case search where we use the same filtering and division strategy as in Lefèvre algorithm, but we introduce a more *regular* algorithm – in the sense that it strongly reduces divergence on GPU – in order to compute a lower bound on $\{b - a \cdot x \mod 1 \mid x < \#_p D_i\}$. Hereafter, we will refer to this new algorithm as the regular HR-case search.

In this new HR-case search described in Algorithm 2, we only rely on Property 2 in order to use only the division-based Euclidean algorithm. Instead of testing the length of the interval containing $\{b\}$ like in Lefèvre HR-case search, we test here which length is reduced as in the classical Euclidean algorithm, and then we reduce it and update d accordingly. Let (p, q) be a two-length configuration.

- If $p < q$, we go directly to the configuration (p, r) with $r = q - \lfloor q/p \rfloor \cdot p$.
    - If $\{b\}$ was in an interval of length p: no point is added in the interval containing $\{b\}$ (as $d < p$, $d = d \mod p$).
    - If $\{b\}$ was in an interval of length q: points are potentially added to the left of $\{b\}$. According to Property 3, intervals of length q are split by adding points from the left. Hence the distance d is updated by reduction modulo p.
- If $q < p$, we go directly to the configuration (r, q) with $r = p - \lfloor p/q \rfloor \cdot q$.
    - If $\{b\}$ was in an interval of length q: no point is added in the interval containing $\{b\}$.
    - If $\{b\}$ was in an interval of length p: points are potentially added to the left of $\{b\}$. According to Property 3, intervals of length p are split by adding points from the right. Then the distance d is updated if $d > r$ by reducing $d - r$ modulo q.

In each branch, we reduce the interval of the longest length by using Property 2 and then we update d. In the *else* branch of Algorithm 2, testing if $\{b\}$ was in an interval of length q is useless. Indeed if $d > r$ and $\{b\}$ was in an interval of length q, subtracting r at line 14 would still have been done in the next loop iteration at line 8.

Algorithm 2: New regular lower bound computation and test algorithm.

```
   input : $b - a \cdot x$, $\epsilon''$, $N$
1  initialisation: $p \leftarrow \{a\}$;  $q \leftarrow 1$;  $d \leftarrow \{b\}$;
                   $u \leftarrow 1$;  $v \leftarrow 1$;
2  if $d < \epsilon''$ then return Failure;
3  while True do
4      if $p < q$ then
5          $k = \lfloor q/p \rfloor$;
6          $q = q - k * p$;
7          $u = u + k * v$;
8          $d = d \mod p$;
9      else
10         $k = \lfloor p/q \rfloor$;
11         $p = p - k * q$;
12         $v = v + k * u$;
13         if $d \geq p$ then
14             $d = d - p$;
15             $d = d \mod q$;
16     if $u + v \geq N$ then return $d > \epsilon''$;
```

In practice, the remainders of the continued fraction expansion are computed like in Lefèvre HR-case search with a subtractive division, a division instruction or the hybrid approach.

4.3 Deployment on GPU

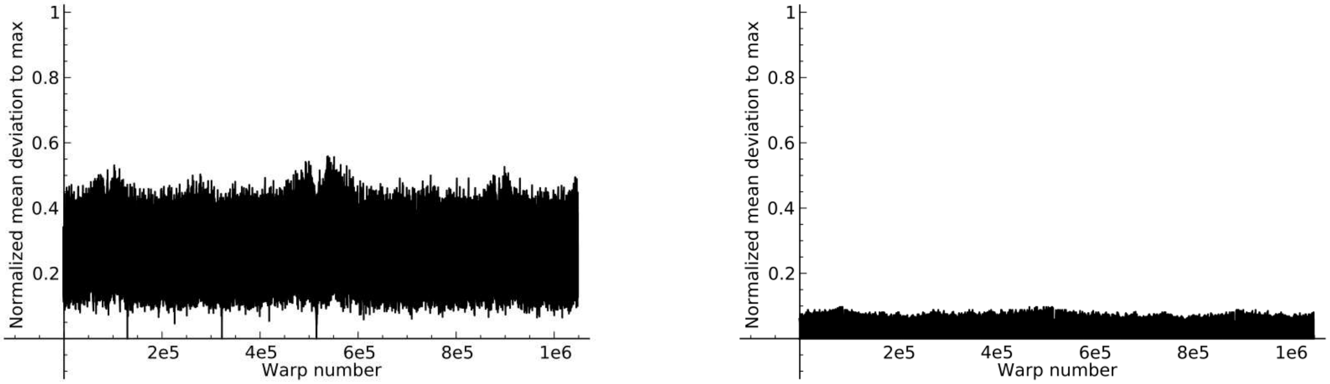
The exhaustive search algorithm perfectly takes advantage of the GPU massive parallelism and of its (partial) SIMD execution. Hence, we will focus on the deployment of the lower bound computation. In this section we present how we have deployed Lefèvre HR-case search and the new regular HR-case search on GPU by reducing divergence at three levels: the filtering strategy, the main loop and the main conditional statement. More details about Lefèvre HR-case search deployment on GPU can be found in [8].

For these deployments, we first changed the data layout to a "structure of arrays" in order to have coalesced memory accesses [14, Sect. 6.2.1]. We also avoided as much as possible consecutive dependent instructions in order to increase the instruction-level parallelism within each thread.

Throughout this section, we will consider the example interval $[1, 1{+}2^{-13}[$ in the binade $[1, 2[$ for the exponential function in double precision, as this binade is considered in [21] as the general case.

4.3.1 Filtering strategy divergence

As a consequence of the filtering strategy, we will have few threads executing phase 2, and fewer executing



(a) Lefèvre HR-case search with specific instructions

(b) Regular HR-case search

Figure 6: Normalized mean deviation to the maximum of the number of main loop iterations per warp among the 2^{20} warps required for the *exp* function in the interval $[1; 1 + 2^{-13}]$.

Phase	Number of intervals	
	Lefèvre	Regular
1	$2^{40} \approx 1.1 \cdot 10^{12}$	$2^{40} \approx 1.1 \cdot 10^{12}$
2	$\approx 3.6 \cdot 10^{9}$	$\approx 1.8 \cdot 10^{10}$
3	$\approx 8.9 \cdot 10^{6}$	$\approx 5.9 \cdot 10^{7}$
HR-cases	243	243

Table 1: Details of argument filtering during HR-case search in $[1, 1 + 2^{-13}]$.

HR-case search	min iteration number	max iteration number	mean iteration number	mean NMDM
Lefèvre	5	328	24	25.6%
With specific instructions	5	31	16	25.7%
Regular	8	19	12	0.1%

Table 2: Comparison of the main loop behavior among the 2^{20} warps required for the different HR-case searches on *exp* function in interval $[1, 1 + 2^{-13}]$.

phase 3. Table 1 shows the number of intervals involved in each phase for an interval D_i containing 2^{40} floating-point numbers. As we can see, very few intervals lead to the exhaustive search step. Hence, executing one kernel computing the three phases leads to an important divergence as we have fewer and fewer active threads within each warp from one phase to the next [8].

To tackle this problem, we propose to use three kernels, one for each phase. This allows us to re-build the grid of threads between each phase, and to run the exact number of threads required by each phase. However, this implies two additional costs.

First, we have to write failing intervals[2] of phase 1 and 2 in consecutive memory locations as we prepare coalesced reads for the next phase. In [8], this was done with atomic operations on the GPU global memory since we had few failing intervals. For some specific binades, the number of failing intervals can be much more important and the numerous atomic operations can then lower the performance. Hence, we use atomic operations on the GPU global memory or compact operations based on parallel prefix sums provided by CUDPP [25], depending on the expected number of failing intervals.

Second, between two phases, we have to transfer back to CPU the number of failing intervals to compute on CPU the optimal grid size for the next phase. This optimal grid size is computed by factoring the number of intervals involved in the next phase, which enables us to minimize the number of useless threads.

It can be noticed in Table 1 that Lefèvre HR-case lower bound computation filters a little more than the new algorithm. Lefèvre HR-case lower bound computation uses indeed subtraction-based Euclidean algorithm when splitting the interval containing $\{b\}$, which results in a number of considered arguments less than $2 \cdot \#_p D_i$. As we always use the division-based Euclidean algorithm in the regular HR-case search, we consider $k \cdot \#_p D_i$ where k is the last computed quotient. However, the geometric mean of the quotients of the continued fraction of almost all real numbers equals Khinchin's constant (≈ 2.69). Hence, we can hope considering $3.69 \cdot \#_p D_i$ arguments with the regular HR-case search.

4.3.2 Loop divergence

The second source of divergence is the main unconditional loop (see line 3 in Algorithms 1 and 2). Fig. 6 shows the NMDM of the number of loop iterations by warp for the different HR-case searches on the testing interval containing 2^{40} double-precision floating-point arguments. Table 2 summarizes statistical informations on the NMDM and the number of iterations for both Lefèvre and the regular HR-case searches.

For Lefèvre HR-case search, this main unconditional loop is an important source of divergence with a mean NMDM of 25.6%, that is to say, a thread remains idle on average 25.6% of the number of loop iterations executed by its warp. To our knowledge there is no *a priori*

[2] Intervals for which the computed lower bound is less than ϵ'' in algorithms 2 and 1.

information on the number of loop iterations that would enable us to statically reorder the intervals in order to decrease this divergence. We also tried to use software solutions to reduce the impact of the loop divergence [8], [10] to no avail because the computation is very fine-grained.

This divergence in Lefèvre HR-case search is mainly due to the fact that the quotients are entirely or partially computed at each iteration depending on the position of b even with the specific instructions (see Sect. 4.1). Thanks to these specific instructions the pathological cases are avoided (see Table 2) but the mean NMDM is still around 25.6%.

In the new regular HR-case search, the key point is that a quotient of the continued fraction expansion of a is entirely computed at each loop iteration, which is not the case in Lefèvre HR-case search. Hence, the number of loop iterations only depends on the number of quotients of the continued fraction expansion of a computed to reach $\#_p D_i$ points on the segment. As the number of quotient to compute is very close from one interval to the next, we reduce the mean NMDM by warp to 0.1%.

4.3.3 Branch divergence

The third source of divergence is on the main conditional statement on the value d (see line 4 in Algorithms 1 and 2). We aim at reducing the number of instructions controlled by the branch condition, and, if reduced enough, benefit from the GPU branch predication [14, Sect. 9.2]. This branch predication enables indeed, for short sections of divergent code, to fill at best the pipelines by scheduling both *then* and *else* branches for all threads: thank to a per-thread predicate, only the relevant results are actually computed and finally written.

By looking carefully at the content of each branch in Lefèvre HR-case search, we can notice that they contain the same instructions, except that the variables x (respectively u) and y (resp. v) are interchanged, and that x is subtracted to b. We therefore swap the two values x and y (resp. u and v) to remove the common instructions from the conditional scope as described in Algorithm 3. The swap implies a small extra cost but we thus reduce the portion of divergent code, and hence the number of divergent instructions.

To minimize the extra cost of the swap, we swap the values only when this is required, that is to say we swap the values only if the evaluation of the condition $d < p$ changes at line 15 of Algorithm 3. This enables us to minimize the number of swap operations. In practice, each swap is performed thanks to an auxiliary variable.

As far as the new regular HR-case search is concerned, there is in Algorithm 2 as much branch divergence within the unconditional loop as in Algorithm 1. However the main conditional statements of the two algorithms are rather different. In Lefèvre HR-case search, this test depends on the position of the point b at each iteration. In the regular HR-case search, it depends on the length to reduce. Unlike the test on the position of b, the test on the length to reduce is deterministic as the regular HR-case search computes a quotient of the continued fraction expansion of a at each loop iteration. Hence the evaluation of the condition switches at each loop iteration and it first evaluates to *True* as p is initialized to $\{a\}$ and q to 1. Therefore, by unrolling two loop iterations, we can avoid this test which leads to Algorithm 4 where the branch divergence is strongly reduced.

Algorithm 3: Lefèvre's lower bound computation and test algorithm with swap.

```
input : $b - a \cdot x$, $\epsilon''$, $N$
1  initialisation:
     $p \leftarrow \{a\}$;  $q \leftarrow 1 - \{a\}$;  $d \leftarrow \{b\}$;
     $u \leftarrow 1$;  $v \leftarrow 1$;  $are\_swapped \leftarrow False$;
2  if $d < \epsilon''$ then return Failure;
3  if $(d \geq p)$ then
4      SWAP$(p, q)$; SWAP$(u, v)$;
5      $are\_swapped \leftarrow True$;
6  while True do
7      if $are\_swapped$ then
8          $d \leftarrow d - p$;
9          if $d < \epsilon''$ then return Failure;
10     $k = \lfloor q/p \rfloor$;
11     $q \leftarrow q - k * p$; $u \leftarrow u + k * v$;
12     if $u + v \geq N$ then return Success;
13     $p \leftarrow p - q$; $v \leftarrow v + u$;
14     if $are\_swapped$ xor $(d \geq p)$ then
15         SWAP$(p, q)$; SWAP$(u, v)$;
16         $are\_swapped \leftarrow not(are\_swapped)$;
```

Algorithm 4: New regular lower bound computation and test algorithm unrolled.

```
input : $b - ax$, $\epsilon''$, $N$
initialisation:  $p \leftarrow \{a\}$;  $q \leftarrow 1$;  $d \leftarrow \{b\}$;
                 $u \leftarrow 1$;  $v \leftarrow 1$;
while True do
    $k = \lfloor q/p \rfloor$;
    $q = q - k * p$;
    $u = u + k * v$;
    $d = d \mod p$;
    if $u + v \geq N$ then return $d > \epsilon''$;
    $k = \lfloor p/q \rfloor$;
    $p = p - k * q$;
    $v = v + k * u$;
    if $d \geq p$ then
        $d = d - p$;
        $d = d \mod q$;
    if $u + v \geq N$ then return $d > \epsilon''$;
```

5 Polynomial approximation generation on GPU

In this section, we detail how we have deployed on GPU the generation of polynomial approximations required for the HR-case search algorithms described in Sect. 4.

The principle is to approximate the function f on a large interval by a polynomial P of degree δ with for example a Taylor approximation. Then, to have precise approximations P_i over intervals $[iN; (i+1)N]$ of N arguments, we use Taylor shifts of P that is to say we compute $P_i(x) = P(x + iN)$. The tabulated difference shift [5], [26], [27], the most often used shift in this context, is intrinsically sequential and requires multi-precision arithmetic which is problematic for a deployment on the parallel GPU architecture.

In the following, we will first present a method named *hierarchical* method [5], [26] to change one Taylor shift by N into several Taylor shifts by 1. Then, we present two existing Taylor shift algorithms:

- the tabulated difference shift which sequentially iterates a shift of the polynomial P_i to obtain P_{i+1} with only multi-precision additions;
- and the straightforward shift which computes the P_i's from P in parallel but requires multi-precision multiplications and additions.

Finally we will discuss how the two shifts can be combined in an hybrid CPU-GPU Taylor shift algorithm which requires only fixed size multi-precision addition on GPU.

More details on these algorithms and their error propagation can be found in [5], [27]. There exist asymptotically more efficient algorithms and an overview can be found in [28].

5.1 Taylor shift algorithms

We first describe a hierarchical method originally described in [5], [26] which transforms one shift by N of a polynomial of degree δ into $\delta+1$ shifts by 1. This method requires the input polynomial to be represented in the binomial basis. Hence, we define the forward difference operator and its application to interpolate a polynomial in the binomial basis.

Definition 5. *The forward difference operator, denoted Δ_h is defined as $\Delta_h[P](x) = P(x+h) - P(x)$. We write Δ_h^i the composition i times of Δ_h and $\Delta = \Delta_1$.*

Using this forward difference operator, one can get an efficient algorithm to interpolate a polynomial of degree δ in the binomial basis, given the values $\{P(i), 0 \le i \le \delta\}$ as

$$P(x) = \sum_{i=0}^{\delta} \Delta^i[P](0) \cdot \binom{x}{i}.$$

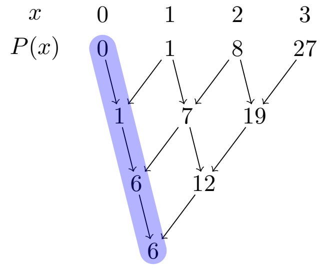


Figure 7: Newton interpolation of polynomial x^3.

An example is shown in Fig. 7. This interpolation is computed using the definition of Δ and as initial values $\Delta^0[P](x) = P(x)$. This algorithm is similar to the Newton interpolation with the forward difference operator used instead of the forward divided difference operator.

Now, we describe the hierarchical method. Given a polynomial P, we want to build a scheme to shift this polynomial in consecutive arguments following an arithmetic progression with common difference N. Hence, if we consider the univariate polynomial P as a bivariate polynomial such that $P(x) = P(kN + m)$, by interpolation in binomial basis with respect to the variable m, we obtain

$$P(kN + m) = \sum_{j=0}^{\delta} a_j(k) \binom{m}{j}.$$

Hence, by evaluating the $a_j(k)$ in i we get $P_i(m)$ in the binomial basis.

To obtain all the P_i, we have to compute the consecutive evaluations of $a_j(k)$. This can be performed efficiently with the *tabulated difference* shift [26], [27]. According to the forward difference operator definition,

$$\Delta^i[P](x) = \Delta^{i-1}[P](x+1) - \Delta^{i-1}[P](x),$$

that is to say

$$\Delta^{i-1}[P](x+1) = \Delta^{i-1}[P](x) + \Delta^i[P](x).$$

Furthermore, if P is of degree δ,

$$\Delta^\delta[P](x) = \Delta^\delta[P](x+1) = \cdots = \Delta^\delta[P](x+m)$$

for any integer $m > 0$, and is constant as it is the δ^{th} discrete derivative of P times $\delta!$. An illustration of this algorithm can be found in Fig. 8. Hence, the only needed operations to obtain the consecutive evaluations of the polynomials $a_j(k)$ are multi-precision additions of the coefficients.

Obtaining the consecutive evaluations of $a_j(k)$ can also be performed with the *straightforward shift*. This algorithm multiplies the vectors of $a_j(k)$ coefficients by a triangular matrix constructed using Newton's binomial theorem. If we consider $a_j(k)$ as polynomials in binomial basis, the triangular matrix is

$$\begin{pmatrix} \binom{k}{0} & \binom{k}{1} & \cdots & \binom{k}{\delta} \\ 0 & \binom{k}{0} & \cdots & \binom{k}{\delta-1} \\ 0 & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \binom{k}{0} \end{pmatrix}.$$

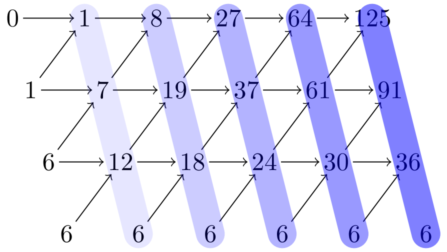


Figure 8: Tabulated difference shift for evaluating the polynomial x^3.

This multiplication exactly corresponds to applying k times the tabulated difference algorithm. Furthermore, this matrix is Toeplitz, which can be used to speed up the matrix-vector multiplication for high degree. Hence, only the first $\binom{kN}{i}$ with $0 \leq i < \delta + 1$ are needed to compute its coefficients.

5.2 Hybrid CPU-GPU deployment

In our deployment, we use the hierarchical method as described in [5], [26]. The polynomial P is a Taylor polynomial of degree 2 approximating the targeted function like in [26]. We interpolate it in the binomial basis using the hierarchical method with $N = 2^{15}$ as we want to use the Boolean tests described in Sect. 4 on intervals containing 2^{15} arguments (see [5] for more explanations). Hence we have

$$P_k(x) = P(k2^{15} + m) = \sum_{j=0}^{2} a_j(k)\binom{m}{i}.$$

As the interpolation in the binomial basis is done once, it is precomputed on CPU.

Hence, to obtain all the P_i, we have to deploy on GPU the computation of the consecutive evaluations of $a_j(k)$.

On one hand, the tabulated difference shift is efficient as it requires only multi-precision additions. This method is thus used in the reference CPU implementation [26]. However this is an intrinsically sequential method, which prohibits its direct deployment on GPU.

On the other hand, the straightforward shift is embarrassingly parallel, but requires multi-precision multiplications and divisions to compute the binomials and multi-precision multiplications and additions to compute the matrix-vector products.

In order to benefit from the efficiency of the tabulated difference shift on GPU, we therefore use an hybrid strategy that relies on both the CPU and the GPU: we compute shifts of $a_j(k)$ by $tS + s$ with $0 < t < T$ and $0 < s < S$. We vary t and shift by tS sequentially on CPU with the straightforward shift to form packets[3]. This way, we generate T packets of size S. All the multi-precision operations on CPU are computed efficiently using the GMP library [29].

Then the $a_j(tS)$ with $0 < t < T$ are transferred to GPU. We run a CUDA kernel of T threads wherein each thread of ID t processes $a_j(tS)$ and computes $a_j(tS + s)$ with $0 < s < S$ using the tabulated difference shift.

Furthermore, as there are $\delta + 1$ independent a_j polynomials ($\delta = 2$ in practice), we can run one kernel per a_j and overlap the GPU computation for the polynomial a_j with the CPU computation of the polynomial a_{j+1}.

The only algorithm deployed on GPU is therefore the tabulated difference shift which is sequential within each GPU thread, but performed concurrently by multiple threads on multiple polynomials.

As the coefficients of the considered polynomials are large, we need multi-precision addition on GPU. Here only fixed size multi-precision additions are required as bounds on the required precision, depending on the targeted function and exponent of the targeted domain, can be computed before compile time [5], [27]. Multi-precision libraries on GPU [30], [31] have been very recently developed. However, we preferred to have our own implementation of this operation for two main reasons: to use PTX (NVIDIA assembly language) [32] and the *addc* instruction in order to have an efficient carry propagation; and to benefit from the fixed size of the multi-precision words at compile time in order to unroll inner loops. As the *addc* instruction operates only on 32-bit words, multi-precision words are arrays of 32-bit chunks. The multi-precision addition function is implemented as a C++ template with the size of the multi-precision words given as a parameter, which enables an automatic generation of addition functions for each size of fixed multi-precision word required by each binade. As a consequence, the inner loop on the number of chunks can easily be unrolled as the number of loop iterations is known at compile time. Furthermore, in order to have coalesced memory accesses, the word chunks are interleaved in global memory and loaded chunk by chunk in registers.

Finally, it can be noticed that this algorithm is completely regular: there is therefore no divergence issue among the GPU threads here.

6 Performance results

In this section we present the performance analysis of our different deployments. All results are obtained on a server composed of one Intel Xeon X5650 hexcore processor running at 2.67 GHz, one NVIDIA Fermi C2070 GPU and 48 GB of DDR3 memory.

We compare three implementations. The first one is the sequential implementation (named *Seq.*) which is Lefèvre reference code provided by V. Lefèvre. The second one is the parallel implementation on CPU (referred to as *MPI*) which is the sequential implementation with an MPI layer (OpenMPI version 1.4.3) to distribute equally the 8192 intervals composing a binade among the available CPU cores (6 physical cores here). We use a cyclic decomposition which offers a better load balancing than

[3] This computation on CPU could thus be parallelized but the corresponding computation times are minority in practice.

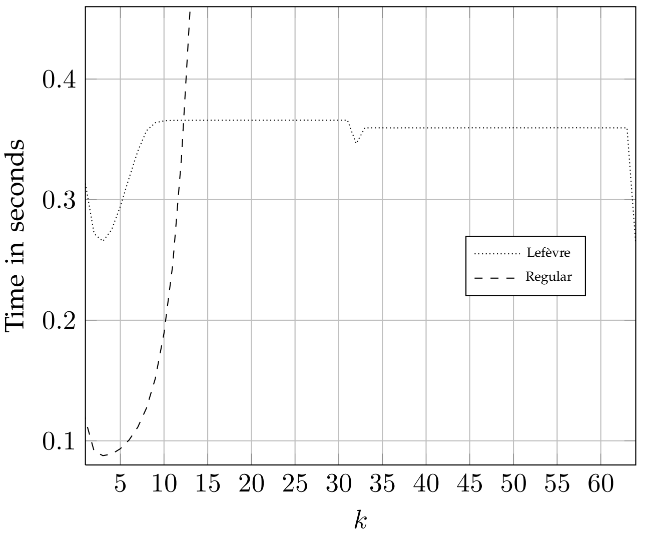


Figure 9: Execution time of the two HR-case searches in the interval $[1 + 2^{-13}]$ according to k threshold on a C2070 GPU.

a block decomposition and run 12 MPI processes to take advantage of the two-way SMT (Simultaneous Multithreading or Hyper-threading for Intel) of each core. The third implementation (named *CPU-GPU*) relies on the GPU and CPU-GPU deployments presented in this paper. The implementations have been compiled with gcc-4.4.5 for CPU code and nvcc (CUDA 4.1) for GPU code.

All the following timings are obtained for searching $(64, 2^{-32})$ HR-cases of *exp* function, that is to say extended precision floating-point arguments for which 32 extra bits of precision during evaluation do not suffice to guarantee correct rounding. The measures include all computations and data transfers between the GPU and the CPU.

6.1 HR-case search

We first want to find for Lefèvre HR-case search and for the regular HR-case search, which division version is the fastest one on GPU among the subtractive division, the hybrid division or the division instruction presented in Sect. 15. We varied the parameter k, referring to the threshold on the size of the quotient, from 0 (division instruction) to 64 (subtractive division as we use 64-bit integers).

As shown in Fig. 9, on a C2070 GPU the optimal k is 3 for both HR-case searches. We therefore use the hybrid division with $k = 3$ in all the following testings. This result is similar to the one observed by Lefèvre [21], which had an optimal k of 3 on CPU. We also run the same test on our X5650 CPU and obtain an optimal k of 3 for Lefèvre HR-case search and for the regular HR-case search. This means that the ratio $\frac{C_{div}}{C_{sub}}$ mentioned in Sect. 15 is roughly the same on CPU and on GPU. Also, this value is small partly due to the expected small values of the quotients of the continued fraction expansion (see Sect. 4.1).

	Seq.	MPI	CPU-GPU	$\frac{\text{Seq.}}{\text{MPI}}$	$\frac{\text{Seq.}}{\text{CPU-GPU}}$
Pol. approx.	43300.81	5251.53	788.84	8.25	54.89
Lefèvre	36816.10	5292.67	2446.27	6.96	15.05
Regular	34039.94	4716.97	711.92	7.22	47.81
Lef. /Reg.	1.08	1.12	3.44		

Table 3: Timings comparison (in seconds) of different implementations of the polynomial approximation generation and of Lefèvre and regular HR-case search in $[1, 2[$.

Furthermore, we notice that the new regular HR-case search is more sensitive to this parameter than Lefèvre HR-case search (it achieves nearly 300 seconds when $k > 30$). There can be two explanations to this behavior. First, d is reduced only by subtractions in Lefèvre HR-case search whereas it is reduced by divisions in the regular HR-case search. This also implies that when a large partial quotient is encountered in the continued fraction expansion, another large quotient is potentially computed to reduce b. Second, as Lefèvre computes each quotient by a first subtraction and then by a division, quotients equals to 1 do not need a division. As a quotient equals 1 with probability 0.4 during the continued fraction expansion much fewer divisions are computed in Lefèvre HR-case search.

We also searched for the optimal block sizes on GPU and tried to increase the number of intervals computed per thread in every GPU kernel, in order to optimize occupancy and computation granularity. However increasing the number of intervals per thread do not improve performances since the occupancy of each kernel is already high enough.

Now we show in Table 3 performance results of the HR-case search over the binade $[1, 2[$ as it corresponds to the general case according to [21]. First, we remark that Lefèvre and regular HR-case searches take advantage of the two-way SMT on the multi-core tests as we have a parallel speedup higher than the number of cores. Then, the deployment of Lefèvre HR-case search on GPU offers a good speedup of 15.05x over one CPU core and 2.16x over six cores. Finally, the new regular HR-case search delivers over Lefèvre HR-case search a slight gain of 8% on one CPU core, of 12% over six CPU cores and an important speedup of 3.44x on GPU. This result in a very good speedup of 51.71x for the regular HR-case search on GPU over Lefèvre HR-case search on one CPU core, and of 7.43x over six CPU cores.

6.2 Polynomial approximation generation

We show in Table 3 performance results of the polynomial approximation generation step over the binade $[1, 2[$.

We first observe that the polynomial approximation generation takes great advantage of SMT with a speedup

	MPI		CPU-GPU		MPI Lef./CPU-GPU	
	$[1,2[$	$[128,256[$	$[1,2[$	$[128,256[$	$[1,2[$	$[128,256[$
Polynomial approximation generation	5336.81	11243.26	785.14	1612.03	6.74	6.97
Lefèvre HR-case search	5292.67	169911.90	2446.78	51530.44	2.16	3.30
Regular HR-case search	4716.96	–	711.8	61581.87	7.44	2.76

Table 4: Timings (in seconds) for binades $[1,2[$ and $[128,256[$. Timings for the MPI regular HR-case search over $[128,256[$ have been omitted because they are prohibitive.

of 8.25x when using six CPU cores. This is mainly due to the high latency caused by the carry propagation during the multi-precision addition which can be partly offset by the SMT execution. Concerning the CPU-GPU deployment of the polynomial approximation generation, the times includes the CPU computations, the data transfers from CPU to GPU and the GPU computation. This hybrid CPU-GPU deployment greatly takes advantage of the GPU as all the threads perform independent computations and as the control flow is perfectly regular among the GPU threads. It offers thus a speedup of 54.89x over the one CPU core execution and of 6.66x over the six core execution.

	Lefèvre		Regular	
	Arguments	Time (s)	Arguments	Time (s)
Phase 1	$9.01 \cdot 10^{15}$	2372.60	$9.01 \cdot 10^{15}$	583.97
Phase 2	$3.19 \cdot 10^{13}$	61.31	$1.62 \cdot 10^{14}$	91.41
Phase 3	$7.65 \cdot 10^{10}$	11.02	$5.14 \cdot 10^{11}$	35.17

Table 5: Details on each phase for Lefèvre and regular HR-case searches on GPU in the binade $[1,2[$.

	Lefèvre		Regular	
	Arguments	Time (s)	Arguments	Time (s)
Phase 1	$9.01 \cdot 10^{15}$	4097.41	$9.01 \cdot 10^{15}$	1634.67
Phase 2	$8.97 \cdot 10^{15}$	30003.78	$9.01 \cdot 10^{15}$	21443.58
Phase 3	$4.19 \cdot 10^{14}$	17428.16	$9.00 \cdot 10^{14}$	38480.87

Table 6: Details on each phase for Lefèvre and regular HR-case searches on GPU in the binade $[128,256[$.

6.3 Overall performance results

In this subsection, we present detailed performance results for the overall algorithms on different binades. In the following tables, one can remark that the total times are slightly higher than the sum of the three phases. This is due to the cost of measuring time for each phase.

In Table 4, the timings are obtained over two binades. The binade $[1,2[$ corresponds to the general case according to [21], where the $\exp$ function is well approximated by a polynomial of degree one, and the binade $[128,256[$ corresponds to the last entire binade before overflow, where the $\exp$ function is hard to approximate by a polynomial of degree one.

We can first observe that speedups on GPU-CPU over CPU of the polynomial approximation generation are similar, even if in the binade $[128,256[$ we use longer multi-precision words (maximum coefficient sizes are 320 bits for the binade $[1,2[$ and 448 for the binade $[128,256[$) and polynomials of higher degree ($\max_j(\deg a_j(x))$ is 6 for the binade $[1,2[$ and 10 for the binade $[128,256[$).

It has to be noticed that the HR-case search is much slower in the binade $[128,256[$ than in the binade $[1,2[$ (22.7x and 86.5x for Lefèvre and regular HR-case searches respectively on GPU). Moreover, Lefèvre HR-case search delivers a better speedup on GPU over CPU in $[128,256[$ compared to $[1,2[$, and regular HR-case search delivers a lower speedup.

The high computation times required in the binade $[128,256[$ and the disparities in speedups of Lefèvre and regular HR-case searches can be explained by the truncation error ϵ_{trunc} introduced by the Boolean tests used in both filtering strategies.

We therefore present in Table 5 the filtering and timing details of the Lefèvre and the regular HR-case searches over the binade $[1,2[$. In Table 5, both HR-case searches split the entering intervals into 8 sub-intervals in phase 2. In this binade the $\exp$ function is well approximated by a polynomial of degree one. This implies that the error ϵ_{trunc} due to the truncation to degree one is low compared to the error ϵ we want to test, and the Boolean tests used in Lefèvre and regular HR-case searches fail rarely.

However, as stated in Sect. 4.3.1, the new regular HR-case search filters less intervals than Lefèvre HR-case search. This increases the amount of time spent in phases 2 and 3 by a factor 1.49 and 3.19 respectively. However, we can observe a good speedup of 4.06x in phase 1 due to the regularity of the new regular HR-case search. As phases 2 and 3 are minority, the new regular HR-case search offers a total speedup of 3.44x over Lefèvre HR-case search.

Table 6 details the corresponding results for the binade $[128,256[$ where the $\exp$ function is hard to approximate by a polynomial of degree one. This implies that ϵ_{trunc} is high compared to ϵ, and the Boolean tests used in Lefèvre and regular HR-case searches fail very often. We set the Lefèvre HR-case search to split the entering intervals into 16 sub-intervals in phase 2 and the regular HR-case search to split the entering intervals into 32 sub-intervals in phase 2. This is due to the need of balancing phase 2 and phase 3 in order to obtain the best performance. Here, only the regular HR-case search uses parallel prefix sums for the compaction operation between each phase.

As the Boolean tests fail very often, the critical phases for this binade are the phases 2 and 3. Hence, Lefèvre HR-case search is more efficient as it filters more than the new regular HR-case search. In this binade, 10.00%

of the initial arguments are involved in phase 3 with the regular HR-case search against 4.65% with Lefèvre HR-case search. This results in Lefèvre HR-case search being 16.3% faster than the regular HR-case search on GPU for this binade. Moreover, as phase 3 corresponds to the exhaustive search, which is embarrassingly parallel and which offers a completely regular control flow, we still have a good speedup on GPU (up to 3.30x with Lefèvre HR-case search over a hex-core CPU).

Hence, both HR-case searches can be used depending on the truncation error. This truncation error directly depends on the coefficient of the term of degree two of the polynomial approximation. A threshold on the truncation error to switch from one HR-case search to the other can be precomputed. One can also use the ratio of the number of intervals in phase 3 over the number of intervals in phase 1 of the previous interval to select the appropriate HR-case search algorithm for the current interval. As shown in Table 4, this let us benefit from a very good speedup of 7.44x on a GPU over a hex-core CPU when the function is well approximated by a polynomial of degree one, and from a good speedup of 3.30x otherwise. For example, with the exponential function in double precision, out of twenty-two binades which do not evaluate to overflow or underflow, fifteen binades are more efficiently computed using the new regular algorithm.

However, both HR-case searches are slow when the truncation error ϵ_{trunc} is high compared to the targeted error ϵ. The best here should be to consider a Boolean test using polynomials of higher degree like in the SLZ algorithm.

7 Conclusion and future work

In this paper, we have proposed a new algorithm based on continued fraction expansion for HR-case search which improves Lefèvre HR-case search algorithm by strongly reducing loop and branch divergence, which is a problem inherent to GPU because of their partial SIMD architecture. We have also proposed an efficient deployment on GPU of these two HR-case search algorithms and an hybrid CPU-GPU deployment for the generation of polynomial approximations.

When searching for HR-cases of the $\exp$ function in double precision, these deployments enable an overall speedup of up to 53.4x on one NVIDIA C2070 GPU over a sequential execution on one Intel X5650 CPU core, and a speedup of up to 7.1x on one NVIDIA C2070 GPU over one Intel X5650 hex-core CPU.

In the future, we plan to investigate whether our new regular HR-case search can benefit from other SIMD architectures like vector units (SSE, AVX, ...) on multi-core CPU and MIC (Many Integrated Cores) architectures. This will require an OpenCL [13] implementation and an effective automatic vectorization by the OpenCL compiler.

We also hope to further decrease the computation times by deploying on GPU the SLZ algorithm which tests the existence of HR-cases with higher degree polynomials. This algorithm heavily relies on the use of the LLL algorithm. The deployment of this algorithm on GPU is therefore far from trivial if one wants to obtain good performance. Porting the LLL algorithm to GPU will be the next step of this work.

8 Acknowledgement

This work was supported by the TaMaDi project of the french ANR (grant ANR 2010 BLAN 0203 01). The authors want to thank Vincent Lefèvre for helpful discussions on the HR-case search and for providing us with his implementations and results. We also thank Polytech Paris-UPMC and their system administrator team for allowing us to use their CPU-GPU server.

References

[1] Microprocessor Standards Commitee, "IEEE Standard for Floating-Point Arithmetic," tech. rep., IEEE Computer Society, August 2008.

[2] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-point Arithmetic*. Birkhauser, 2009.

[3] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Trans. Math. Softw.*, vol. 17, pp. 410–423, September 1991.

[4] V. Lefèvre, "An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$," Tech. Rep. 97-18, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, 1997.

[5] V. Lefèvre, *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École normale supérieure de Lyon, 2000.

[6] D. Stehlé, V. Lefèvre, and P. Zimmermann, "Worst cases and lattice reduction," in *16th IEEE Symposium on Computer Arithmetic (ARITH 16)*, pp. 142–147, IEEE Computer Society, 2003.

[7] D. Stehlé, V. Lefèvre, and P. Zimmermann, "Searching worst cases of a one-variable function using lattice reduction," *IEEE Transactions on Computers*, vol. 54, pp. 340–346, 2005.

[8] P. Fortin, M. Gouicem, and S. Graillat, "Towards solving the table maker's dilemma on GPU," in *Proceedings of the 20th International Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP'2012, pp. 407 – 415, IEEE Computer Society, February 2012.

[9] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," in *International Symposium on Computer Architecture*, ISCA, 2012.

[10] S. Frey, G. Reina, and T. Ertl, "SIMT microscheduling: Reducing thread stalling in divergent iterative algorithms," in *Proceedings of the 20th International Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP'2012, pp. 399–406, IEEE Computer Society, February 2012.

[11] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pp. 3:1–3:8, ACM, 2011.

[12] NVIDIA, *CUDA C Programming Guide, version 4.1*, November 2011.

[13] Khronos Group, *The OpenCL Specification Version 1.2*, November 2011.

[14] NVIDIA, *CUDA C Best Practices Guide, version 4.1*, January 2012.

[15] G. Hanrot, V. Lefèvre, D. Stehlé, and P. Zimmermann, "Worst cases of a periodic function for large arguments," in *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, ARITH 18, pp. 133–140, 2007.

[16] N. B. Slater, "The distribution of the integers n for which $\{\theta n\} < \phi$," *Proceedings of the Cambridge Philosophical Society*, vol. 46, pp. 525–534, October 1950.

[17] V. T. Sõs, "On the theory of diophantine approximations I," *Acta Math. Acad. Sci. Hungar.*, vol. 8, pp. 461–472, 1957.

[18] V. T. Sõs, "On the distribution mod 1 of the sequence nα," *Ann. Univ. Sci. Budapest*, vol. 1, pp. 127–134, 1958.

[19] T. Van Ravenstein, “The three gap theorem (Steinhaus conjecture),” *Australian Mathematical Society*, vol. Series A, no. 45, pp. 360–370, 1988.

[20] N. B. Slater, “Gaps and steps for the sequence $n\theta \mod 1$,” *Mathematical Proceedings of the Cambridge Philosophical Society*, pp. 1115–1123, 1967.

[21] V. Lefèvre, “New results on the distance between a segment and $\mathbb{Z}^2$. application to the exact rounding,” in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ARITH 17, pp. 68–75, IEEE Computer Society, 2005.

[22] R. O. Kuzmin, “Sur un problème de Gauss,” in *Atti del Congresso Internationale dei Matematici*, vol. 6, pp. 83–89, 1932.

[23] A. Y. Khinchin, *Continued fractions*. Dover, 1997.

[24] D. E. Knuth, *The Art of Computer Programming*, vol. 2 (Seminumerical Algorithms). Addison-Wesley, second ed., 1981.

[25] S. Sengupta, M. Harris, and M. Garland, “Efficient parallel scan algorithms for GPUs,” Tech. Rep. NVR-2008-003, NVIDIA, December 2008.

[26] V. Lefèvre, J.-M. Muller, and A. Tisserand, “Toward correctly rounded transcendentals,” *IEEE Transactions on Computers*, vol. 47, no. 11, pp. 1235–1243, 1998.

[27] F. de Dinechin, J.-M. Muller, B. Pasca, and A. Plesco, “An FPGA architecture for solving the Table Maker’s Dilemma,” in 22^{nd} *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, ASAP 2011, pp. 187–194, IEEE Computer Society, 2011.

[28] J. von zur Gathen and J. Gerhard, “Fast algorithms for Taylor shifts and certain difference equations,” in *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, ISSAC ’97, pp. 40–47, ACM, 1997.

[29] T. Granlund and the GMP development team, *GNU MP*, 4.3.2 ed., January 2010.

[30] T. Nakayama and D. Takahashi, “Implementation of multiple-precision floating-point arithmetic library for GPU computing,” in *Proceedings of the 23rd IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS 2011, pp. 343–349, December 2011.

[31] M. Lu, B. He, and Q. Luo, “Supporting extended precision on graphics processors,” in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN ’10, pp. 19–26, ACM, 2010.

[32] NVIDIA, *Parallel thread execution ISA Version 3.0*, February 2012.